Relative Debugging of Automatically Parallelized Programs*

Gabriele Jost Robert Hood

Computer Sciences Corporation
NASA Advanced Supercomputing Division
NASA Ames Research Center
{gjost,rhood}@nas.nasa.gov

Abstract

We describe a system that simplifies the process of debugging programs produced by computer-aided parallelization tools. The system uses relative debugging techniques to compare serial and parallel executions in order to show where the computations begin to differ. If the original serial code is correct, errors due to parallelization will be isolated by the comparison.

One of the primary goals of the system is to minimize the effort required of the user. To that end, the debugging system uses information produced by the parallelization tool to drive the comparison process. In particular, the debugging system relies on the parallelization tool to provide information about where variables may have been modified and how arrays are distributed across multiple processes. User effort is also reduced through the use of dynamic instrumentation. This allows us to modify the program execution without changing the way the user builds the executable.The use of dynamic instrumentation also permits us to compare the executions in a fine-grained fashion and only involve the debugger when a difference has been detected. This reduces the overhead of executing instrumentation.

1 Background

One of the problems facing scientific programmers on high-end computers is that as performance requirements drive up the complexity of machines, they also drive up the complexity of programming models used on them. As a consequence, debugging such codes becomes more difficult. In this paper we describe how automated debugging support can alleviate some of those problems. We begin by providing some background on the target machines and programming model that we are addressing.

1.1 Programming Distributed Memory Computers

A common approach for delivering high performance in computers today is to use a distributed memory architecture. Such a computer consists of a number of processors connected together in a network. Each processor has its local memory that it can access directly. Data from other processors must be accessed via the network.

In this paper we consider the SPMD (Single Program/Multiple Data) programming paradigm, where each processor executes the same program on a subset of the total data. Using this paradigm, computations being performed by one process

*This work was supported through NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D.

†The authors' address is NASA Ames Research Center, M/S T27A-1, Moffett Field, CA 94035.

This paper is an extended version of the paper "Support for Debugging Automatically Parallelized Programs" presented by the authors at AADEBUG 2000, Munich, Germany, August 2000.

will often require data calculated on another process, and data has to be moved between the processes. This data movement is typically performed by explicit message passing from one processor to another using a message passing library like MPI [17] or PVM [20]. The development of a parallel program based on message passing adds a new level of complexity to the software engineering process since not only the computation, but also the explicit movement of data between processes must be specified. Given the enormous investment made in existing scientific applications, there is a strong incentive to produce parallel versions through a conversion process rather than re-implementing from scratch.

1.2 Converting Serial Codes to Message Passing Parallel Codes

When converting a sequential program into parallel code, one way to achieve parallelism is to partition the elements of an array among the processors and have each processor update only the array elements that are assigned to it. A straightforward way to convert a serial loop into a parallel loop based on message passing is to distribute the loop iterations among the processors. The array is logically partitioned into chunks, and each processor is assigned one or more of the blocks. The processor is then responsible for updating the array elements assigned to it. For example, the Fortran loop

```
do i = 1, n
    a(i) = b(i) + 2
end do
```

could be parallelized by splitting up arrays `a` and `b` into contiguous sections. Each processor would execute:

```
do i = lower, upper
    a(i) = b(i) + 2
end do
```

where `lower` and `upper` denote the lower and upper index of the array section assigned to the processor. Now consider the loop:

```
do i = 1, n
    a(i) = b(i-1) + 2
end do
```

If array `b` is partitioned the same way as array `a`, processor p will have to access data from processor p-1. Therefore calls to communication routines have to be inserted. Processor p has to send `b(upper)` to processor p+1 and receive `b(lower-1)` from processor p-1:

```
call send(b(upper), 1, real, p+1, ierr)
call receive(b(lower-1), 1, real, p-1, ierr)
do i = lower, upper
    a(i) = b(i-1) + 2
end do
```

The loop:

```
do i = 1, n
    a(i) = (a(i) + a(i-1)) * 0.5
end do
```

can not be executed in parallel since data from iteration `i` is dependent on data from iteration `i-1`.

In order to determine whether a loop can be parallelized and which updates require data from another process, the array indices have to be analyzed in order to detect dependences. The analysis has to be done between individual statements, iterations of a loop, and subroutine calls. The technique of dependence analysis is well understood [21] and has been implemented in compilers for code optimization.

Discovering dependences manually and then inserting the necessary message passing calls is a tedious and error-prone task. There are several systems that assist the user in the task of parallelizing codes. For example, the *CAPTools* system from

the University of Greenwich [7] can take a serial program in Fortran77 and, with some user guidance, turn it into a message passing parallel program. The user's role in this process is fairly modest. While the tool is analyzing the serial code, it may ask the user for additional information in order to perform a more precise dependence analysis. After the analysis is done, the user chooses a distribution for one or more of the arrays. Then, when *CAPTools* is producing the parallel version, it may ask additional questions about the relative values of variables such as "Can N be larger than M?". The result of this process is a message passing version of the code.

1.3 Errors in Automatically Parallelized Codes

There are several reasons why the automatically generated version might produce results that differ from the serial original version:

1. Parallelization may change the order of execution in some loops and lead to numerical discrepancies. For example, performing a sum reduction in a different order could produce different results because of the non-associativity of floating point addition.
2. The serial code may have errors. For example, if the serial code references an undefined variable, execution of the parallel code may produce a different result by reading a different value for the undefined variable.
3. The tool for automatic parallelization may be buggy.
4. The user can introduce errors by providing incorrect inputs to the tool, e.g., incorrect responses to the system's queries or incorrect removal of dependences,

Discrepancies due to reasons 1 and 2 are not specific to parallelization, but can arise from any compiler optimization. Discrepancies due to errors in the tool depend on the maturity of the tool and are hoped to be rare. Discrepancies due to incorrect user inputs can take several forms, as we discuss below. In order to be as specific as possible in these scenarios, we consider user interactions with, and code generated by, the *CAPTools* system. Similar errors are possible with other parallelization tools.

By its nature, the *CAPTools* system, like any parallelization support tool, has to be conservative in its assumption about data dependences. Often the existence of a data dependence will depend on certain input parameters. In such a case parallelization will not be performed in order to assure correctness of the code for all possible input values. The strength of the *CAPTools* system lies in the fact that it allows the user to provide information about values of certain variables. This knowledge leads to a more precise dependence analysis and the generated parallelized code will be highly efficient. The drawback is that this also opens a window to the introduction of errors. Consider the following code fragment as an example:

```
  program linearize
C main routine
  double precision phi2(100*100), phi3(100,100)
  read idim
  ...
  do j = 1, 100
     do i = 1, 100
       phi2(i + j*100*idim) = phi2(i + j*100*idim) +
                  0.5 * (phi3(i-1,j)+phi3(i+1,j)+phi3(i,j-1)+phi3(i,j+1))
     end do
  end do
  call output (phi2, nptsx, nptsy)
  end
```

If `idim` > 0, the `j`-loop can be parallelized. If `idim=0`, the `j`-loop carries a dependence, since `phi2(i)` from iteration `j` is used in iteration `j+1`. The value of `idim` is not known at compile time. By default, *CAPTools* will assume a dependence and not parallelize the loop. However, it is possible for the user to inform *CAPTools* that `idim>0`. If this is not true for certain sets of input data, the produced results will be incorrect. Among the future plans for CAPTools is the insertion of assertions into the generated code to catch these kind of incorrect user assumptions. At the moment, however, this feature does not exist. Also, the addition of assertions might become rather expensive in some situations.

There could also be situations where *CAPTools* does not have access to all of the source code. For example, suppose the source code of subroutine `sub` is not provided, and the user incorrectly says that the statement

```
call sub(a, n)
```

does not modify array `a`. This may result in a parallelized loop where a processor uses stale values for parts of `a` instead of the up-to-date ones residing on another processor.

As a third scenario, consider the loop:

```
do j = 1, n
    a(ind(j)) = a(ind(j)) * 0.5
end do
```

If *CAPTools* has no further information about the array `ind`, it will assume a dependence and not parallelize the loop. The user, however, may have special knowledge that would make the loop parallelizable. For example, in this case he may know that `ind(j)=j`. For these situations *CAPTools* allows the user to explicitly manipulate the results of the CAPTools dependence analysis. Again, this creates the possibility of introducing errors. In section 6 we give a detailed example of this scenario.

A programmer trying to isolate such bugs in the parallel program faces a daunting task. Not only has the serial source code been sprinkled with calls to communication libraries, but the loop structure may have undergone transformations as well. Since the parallelization tool attempted to optimize the communication patterns, the programmer must use sophisticated reasoning to determine whether a processor is in fact using a current or stale value. Figure 1 contains a small example of how serial code is transformed by *CAPTools*.

From the programmer's perspective, rather than attempting to debug the parallel program directly, a more promising approach is to determine where the parallel computation begins to differ from the serial one. This could be done by instrumenting both codes with print statements and examining the outputs, or by running two debugging sessions side-by-side. Both of these approaches have the drawback, however, that the programmer is required to deal with the tool-produced code in the parallel version.

The goal of our work is to provide support for automatically finding bugs in programs parallelized with tools. We feel that such a goal is feasible because we have:

- a reference program (serial code) for determining the expected behavior, and
- mapping information from the parallelization tool that conveys how the serial program was transformed into the parallel one.

This combination permits the debugger to do side-by-side executions of the serial and parallel versions of the code. In partic-

```
program main
real*8 u(0:33, 0:33), v(0:33, 0:33)
call loop (u, v)
end

subroutine loop (upar, vpar)
real*8 upar (0:33, 0:33), vpar (0:33, 0:33)
integer i, j, d1, d2
d1 = 33
d2 = 33
do i = 0, d1
  do j = 0, d2
    upar (i,j) = 0.
    vpar (i,j) = 1.
  end do
end do
do i = 1, 32
  do j= 1, 32
    upar (i,j) = upar(i,j) + 0.25 *
+                           (vpar(i-1,j  ) +
+                            vpar(i+1,j  ) +
+                            vpar(i,  j-1) +
+                            vpar(i,  j+1))
  end do
end do
return
end
```

Original serial code.

```
PROGRAM PARALLELmain
INTEGER CAP_LEFT,CAP_RIGHT
PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
REAL*8 u(0:33,0:33),v(0:33,0:33)
INTEGER CAP_BLu,CAP_BHu
COMMON /CAP_RANGE/CAP_BLu,CAP_BHu
INTEGER CAP_ICOUNT
CALL CAP_INIT
call CAP_SETUPPART(0,33,CAP_BLu,CAP_BHu)
call loop(u,v,CAP_BLu,CAP_BHu)
CALL CAP_FINISH()
END

subroutine loop(upar,vpar,
+                CAP_Lupar,CAP_Hupar)
integer CAP_LEFT,CAP_RIGHT
PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
REAL*8 upar(0:33,0:33),vpar(0:33,0:33)
integer i,j,d1,d2
integer CAP_Lupar,CAP_Hupar
COMMON /CAP_RANGE/CAP_BLu,CAP_BHu
integer CAP_BLu,CAP_BHu
integer CAP_j
d1=33
d2=33
  do i=MAX(0,CAP_Lupar),MIN(d1,CAP_Hupar),1
    do j=0,d2,1
    upar(i,j)=0.
    vpar(i,j)=1.
    enddo
  enddo
  do CAP_j=1,32,1
  CALL CAP_EXCHANGE(vpar(CAP_Hupar+1,
+                        CAP_j),
+                   vpar(CAP_Lupar,CAP_j),
+                   1,3,CAP_RIGHT)
  enddo
  do CAP_j=1,32,1
  CALL CAP_EXCHANGE(vpar(CAP_Lupar-1,
+                        CAP_j),
+       vpar(CAP_Hupar,CAP_j),1,3,CAP_LEFT)
  enddo
  do i=MAX(1,CAP_Lupar),MIN(32,CAP_Hupar),1
    do j=1,32,1
    upar(i,j)=upar(i,j)+0.25*(vpar(i-1,j)+
+       vpar(i+1,j)+vpar(i,j-1)+vpar(i,j+1))
    enddo
  enddo
return
END
```

Output of *CAPTools*.

FIGURE 1. How *CAPTools* transforms a serial loop. The communication calls inserted in the output, such as CAP_EXCHANGE, refer to *CAPTools*-provided routines that are implemented in an appropriate way for the target machine (e.g., in MPI)

ular, the user could compare corresponding states between the two executions without being required to look at the parallel code. In the next section we will discuss possible approaches for automating the execution comparison process.

2 Support for Relative Debugging of Automatically Parallelized Programs

There are many situations in software development where it is helpful to find out how two related programs differ in behavior. One example is that of locating a bug that was introduced between successive versions of a program. *Relative*

debugging [1] is a technique that compares data during execution between a program that produces correct results and one that produces faulty results, to narrow down at which point discrepancies occur.

The technique of relative debugging is directly applicable to the situation of debugging automatically parallelized code since we can assume the existence of a sequential version that produces the correct results. Let us assume we have a sequential program P_s and a parallel program P_p that has been derived from P_s by running its source code through a parallelization tool such as the *CAPTools* program described earlier. If P_p crashes or produces wrong results, we could isolate the bugs by comparing data between P_s and P_p. In doing such a comparison, there are several issues to address.

What data values should be compared between the two executions? A good starting point is a user-specified value that has been determined to be incorrect by examining the results of a previous run. The testing could be made more precise by also comparing values used to define the known incorrect one.

When during execution should they be compared? One possibility would be to perform the comparison immediately after any statement that could change a value of interest. This might be prohibitively expensive in execution time. Another approach is to do a comparison before and after every subroutine execution that could change the value. This effectively brackets the error location to one subroutine. A combination of both methods would first narrow down the discrepancy to a subroutine by coarse-grained comparisons, then re-execute and apply fine-grained comparison within the subroutine.

How do we know if the values are different? Testing equality is something that will vary from application to application. For example, in some programs, scalar values may be considered "the same" if they are within some tolerance. Arrays might only be considered the same if all corresponding elements are equal. Alternatively, it may be acceptable to calculate checksums of arrays and then compare the sums.

How do we get values from multiple address spaces to a place where they can be compared? There are at least three approaches for this.

- One way to perform the comparison debugging would be to manually insert statements to print the array data to a file, recompile, rerun, and then inspect the printed data from the two executables. The drawbacks of this method are obvious, particularly when many processes are involved.
- Another way would be to use an enhanced debugger that controls both executables. We then have the debugger insert breakpoints, compare the data at the breakpoints, and stop when differences are detected. This approach is taken by the GUARD project [1][2][3].
- A third technique is to have the two computations establish communication, transmit and compare their data, and stop when differences are detected. This can be achieved by instrumenting the source code with routines that send or receive data and perform the comparison. This approach was used as early as 1985 at NASA Ames to debug an FFT code that had been ported to a 4-CPU Cray 2 and showed subtle intermittent problems [4]. We have subsequently successfully employed this technique when porting codes to new machine architectures.

How is distributed data handled, as in the case where a distributed array is being compared to a serial analog? If an element-by-element comparison is requested, the distributed array needs to be reconstituted. Thus, array distribution information is required. If checksums are being compared, each process in the parallel computation could calculate a partial checksum. Those values could then be aggregated and compared to the serial checksum.

2.1 The Role of the Parallelization Tool

If a tool is used in the parallelization process, some of the questions of the previous section can be answered without user intervention. Parallelization support tools such as *CAPTools* perform four major steps:

- data dependence analysis across statements, iterations of loops, and subroutine calls,
- partitioning of array data,
- masking calculations (such as distributing loops), and
- generating necessary calls to communication library routines.

If all the information generated in those steps is gathered in a database, the following kind of information is statically available:

- definition-use chains for array elements across statements and subroutines resulting from dependence analysis and
- information about which part of an array belongs to a certain processor, resulting from data partitioning.

The first item of information can be used to identify those functions and subroutines that modify a certain array and should therefore be instrumented for comparison. The second item can be used to determine how a distributed array maps to its serial analog.

2.2 The Role of the Distributed Debugger

In a relative debugging system for tool-parallelized codes, the distributed debugger provides the interface for the user. For example, the user could steer the comparison activities by selecting the arrays that should be compared. In addition, the debugger controls the executions being compared, retrieves information from the parallelization database, and instruments the target programs by having appropriate function calls inserted dynamically into the executables. Finally, the presence of the debugger will permit more extensive state examination and control of execution during the steps taken to isolate the parallelization bugs.

While the technique of relative debugging can be applied to all programming models, the prototype implementation of our debugging system currently only supports SPMD programming. An extension of our system to other programming paradigms is theoretically possible, but would require very extensive mapping information between reference and faulty version of the code.

3 Prototype Implementation

As part of ongoing work in a debugger research project at NASA Ames, we have built a prototype relative debugging system for tool-parallelized codes where we try to minimize the amount of user intervention required. In this section we describe its implementation and in the process discuss how it answers the questions raised in the previous section.

3.1 Determining Locations and Variables to be Compared

Besides being used to produce the message passing program, *CAPTools* also provides vital information to the debugging system. At the heart of *CAPTools* is a dependence analysis system that examines the serial code in order to establish the safety of running loop bodies in parallel. After performing dependence analysis, it transforms the serial code to parallel form, inserting calls to communication libraries, as needed. The results of *CAPTools*'s sophisticated analysis and transformation phases are stored in a database in the file system. This fact makes it possible for a debugger to find out how a serial array was distributed for parallel execution and which routines modify that array [15].

From the dependence analysis database, information about location of statements that assign to a particular array can be obtained. The *CAPTools* developers have provided us with routines that, given any statement in a program and an array name, construct a list of all routines that might define the array value that reaches the statement. In our prototype implementation, the user provides the name of a subroutine and a variable that has been identified as having an unexpected value. For example, if `phi` is a suspicious variable in subroutine `output`, then, by probing the database we might obtain the following information:

```
copy:phi5
update:phi4
setup_grid:phi6
```

This information tells us, that variable `phi5` in routine `copy`, variable `phi4` in routine `update`, and variable `phi6` in routine `setup_grid` might define the suspicious variable `phi`. Therefore, we will perform a comparison of these variables in the corresponding subroutines.

Where the comparisons should be performed depends very much on the desired granularity of the comparisons. For example, a comparison could be performed every time a distributed array is written to. We restrict ourselves to inserting comparison routines on entry and exit of routines that modify the distributed arrays of interest. The restriction is due to limitations in our prototype implementation and will be discussed in Section 7

We would like to add a word of caution regarding the completeness of our instrumentation. As mentioned earlier in Section 1.3, errors in the parallelized code are often due to the fact that the user deletes necessary dependence edges or provides incorrect information regarding the values of certain input parameters. Using this incorrect information may lead to failure to instrument the routine where the error occurs. An example would be the following situation:

```
program test
...
call sub1(a, n)
call sub2(a, n)
...
```

where the error occurs in `sub1`. Let `a` be defined in `sub1` and used in `sub2` and assume that the user has incorrectly deleted the corresponding dependence edge between the two routines. In this case our system will not instrument `sub1` and the error will not be detected at the exit of `sub1`. Instead, the first discrepancy will be indicated at the entry to `sub2`. Although that weakens the result of the relative debugging, the information will still be helpful to the user, particularly in a large application

code. We will discuss the potential use of information about deleted dependence edges in Section 9

In our prototype implementation, we are taking advantage of the fact that *CAPTools* tries to preserve the original program structure as much as possible. For example, no procedure inlining is being performed, and there is a one-to-one correspondence of the subroutine calls that occur in serial and parallel program. *CAPTools* does, however, perform procedure cloning, i.e., in some cases, parallel as well as serial versions of the same procedure exist. We are taking this fact into account by instrumenting the cloned subroutines as well. Loop restructuring that might occur during the parallelization process, such as loop interchange or loop fusion, can make the determination of instrumentation points at statement or loop level difficult, requiring detailed mapping information between the two versions of the code. Due to the coarse level of our instrumentation, which only occurs at entry and exit of subroutines, we do not have to worry about loop restructuring transformations in our current prototype implementation.

Having determined where and what to compare, we also need to address the situation where the suspicious variable is a distributed array. *CAPTools* provides two ways to allocate distributed data. In the replicated data approach, each process allocates the whole array and updates its own part of the data. In the reduced memory approach, each process allocates memory only for the block of data that it is responsible for and possible overlap regions. To perform a comparison we need to know how distributed data from the parallelized program maps onto its undistributed counterpart. Again, we can retrieve this information from the data base with routines provided to us by the *CAPTools* developers. Using the replicated data approach, we obtain the following information when probing the data base, if `phi` is an array declared in subroutine `update`:

```
Information for symbol phi in routine update:
symbol-name: phi
declaration:(16,16)
dimensionality: 2
partition-index: 1
partition-bounds: CAP_LOW_phi, CAP_HIGH_phi
```

This tells us that in subroutine `update`, array `phi` is a 2-dimensional array of size 16x16. It is partitioned in the first dimension. Each process will calculate the partition `phi(CAP_LOW_phi:CAP_HIGH_phi, 1:16)` of array `phi(1:16,1:16)`. In case of 4 processors and blockwise distribution, `CAP_LOW_phi` will be 1, 5, 9, and 13 on processes 1, 2, 3, and 4, respectively. `CAP_HIGH_phi` will be 4, 8, 12, and 16. This knowledge enables us to compare the distributed array from the parallelized code with the corresponding sections of the undistributed array in the serial program. Similar information is available for the reduced memory data allocation scheme.

3.2 Comparing Data Residing in Different Address Spaces

As indicated in Section 2, we perform relative debugging by instrumenting the programs with calls to subroutines that establish communication and perform the comparisons. One of the efficiency concerns we had in our design was avoiding unnecessary copies of data values, especially large arrays. For example, in the case where the serial version of an array needs to be compared with its distributed analog on an element-by-element basis, we don't want to transmit both arrays to a com-

parison agent. Instead, we would prefer to transmit one array to the address space of the other and perform the comparison there.

Our prototype system uses two routines running in the address spaces of the target processes in order to accomplish the comparison of data from different processes.

- One routine resides in the parallelized code and sends to the serial code either the local contributions of a distributed array or the local checksum of a distributed array depending on which way of comparison is selected.
- The receiving routine is in the reference executable. It receives the local contribution from each process of the parallel executable and compares it with the corresponding data of the undistributed array. When a mismatch is detected, a special function is called to indicate that fact.

As mentioned above, we will perform the comparisons at entry and exit of suspicious subroutine calls. If *sub1* and *sub2* are both instrumented routines in the program, the following error scenarios are possible:

- Values correct on entry to *sub1*, wrong on exit from *sub1*: The routine *sub1* has been identified as the culprit and has to be further investigated. The error could also be in an un-instrumented routine called from *sub1*.
- Values correct on entry to *sub1*, wrong on entry to *sub2*: Routines *sub1* and *sub2* are on the same call stack. The error occurred before the call to *sub2*, possibly in an un-instrumented routine. The discrepancy could potentially be due to un-initialized data rather than an error.
- Values correct on exit from *sub1*, wrong on exit from *sub2*: Routines *sub1* and *sub2* are on the same call stack. The error occurred after the return from *sub1*, possibly in an un-instrumented routine.
- Values correct on exit from *sub1*, wrong on entry to *sub2*: Routines *sub1* and *sub2* are not on the same call stack. The error occurred between the two calls, possibly in an un-instrumented routine. Again, the discrepancy could be due to un-initialized data rather than an error.

Another issue concerning comparing of data is determining the particular comparison test to use. Our prototype implementation provides the following alternatives for checking of distributed array data:

- The local checksums of all parts of the distributed array are added and compared to the checksum of the undistributed array. An error is reported if the difference of the checksums exceeds a user supplied threshold. No array distribution information is required to perform this comparison.
- The local checksum of each part of the distributed array is compared to the corresponding partial checksum of the undistributed array. An error is reported if the difference between one of the checksums exceeds a user supplied threshold. This method requires array distribution information so that corresponding array sections can be identified.
- An element-by-element comparison of undistributed and distributed arrays is performed. An error is reported if the difference in one of the elements exceeds a user supplied threshold.This comparison, just as the previous method, requires array distribution information.

We are currently adding up the array elements to build a checksum. By comparing the difference of the checksums to a user supplied threshold we have a mechanism for ignoring small numerical differences. This method has been sufficient for our current needs. For the future we might consider a more sophisticated hashing algorithm as used for example in cryptography.

Such a reduction operation would require bitwise equivalence in the computation rather than numerical equivalence.

3.3 Program Instrumentation

The next question to address is that of how the calls to comparison routines get inserted into the executables. Having the user modify the source is clearly not an option for an automated debugging system. In our prototype we use dynamic instrumentation based on the *DyninstAPI*, a dynamic code adaptation toolset from the University of Maryland [8]. We chose to use dynamic instrumentation for two main reasons:

- We wanted to avoid the context switches that would result from fine-grained instrumentation being executed in the debugger. Such context switches could slow down execution by several orders of magnitude [16].
- We also wanted to minimize what was required of the user. By using dynamic instrumentation we can avoid changes to the compilation process.

The *DyninstAPI* comes as a set of library routines which provide a portable way of inserting new code into a running program. The new code segments can be used to instrument the program in such a way that execution time does not suffer unduly. Besides inserting code segments, the *DyninstAPI* allows operations like:

- attaching to and detaching from a running process,
- inserting or removing subroutine calls from the application program,
- stopping, continuing, and terminating an application program, and
- reading from and writing to areas of memory of the application program.

For example, by using the *DyninstAPI* it is straightforward to patch a running program so that function execution counts are collected. While such a thing is also possible in a conventional debugger, its interpretation of each piece of instrumentation would require several context switches. This can slow down the execution time of some programs by several orders of magnitude. When done using the *DyninstAPI*, the function counts will be collected in the address space of the program itself, and the effect on execution time is minimized.

Another key feature of the *DyninstAPI* is that the interface is analogous to a machine-independent intermediate representation of the instrumentation as an abstract syntax tree. This allows the same instrumentation code to be used on different platforms.

In our relative debugging system, in order to insert calls to the above routines at appropriate points, we use a process that we call the *Instrumentation Server* (IS). It uses the *DyninstAPI* to control and modify the executables. In a *gdb*-like manner, this program accepts commands from standard input. The most important commands are:

- `attach`: attach to a process,
- `createPoint`: create an instrumentation point in a process, and
- `insertCall`: insert a function call at an instrumentation point.

The commands take arguments such as process ID, names of executables, routine names, and specifications of the arguments to be passed to the instrumentation functions. An example is provided in Figure 6.

3.4 User Interface and System Coordination

To coordinate the actions of the various software components involved and to provide an interface to the user, we use *p2d2*, a portable distributed debugger developed at NASA Ames [19]. One of the goals of the *p2d2* project was to build a debugger for distributed programs which was both portable across a variety of target machines and whose user interface scaled to be able to debug at least 256 processes. The result of our work [10] is a debugger that runs on a variety of Unix-based machines and can be used on both MPI and PVM applications. To achieve the portability goal, *p2d2* abstracted serial debugging objects and operations in a service layer. In the current implementation, this "debugger server" is in turn layered on *gdb*, the debugger from the Free Software Foundation [9].

In recent work we extended *p2d2* so that it could provide a global view of distributed data [11]. *P2d2* collects the local data contributions from each processor and assembles a global picture. For this process information about how the array is distributed across the processors is necessary. *P2d2* can obtain this information either from a database, such as the one produced by *CAPTools*, or by having the user provide it via a dialog box.

Getting the components described in Sections 3.1, 3.2, and 3.3 to cooperate to solve the relative debugging problem is the job of *p2d2*. It retrieves the necessary information about critical routines and array distribution information as described above and passes it to the executables via *IS*. Just as in the case of the global array viewer, this information is obtained by probing the *CAPTools* database mentioned earlier. Having retrieved information about where to instrument, *p2d2* then interacts with *IS* to insert the initialization call that provides the distribution information as well as the calls that move the data between processes and perform the comparison.

4 Behind the Scenes of a Relative Debugging Session

To illustrate what activities need to be coordinated, consider the following scenario from the user's perspective. After having used *CAPTools* to parallelize a program *S*, the user runs the resulting code *P* and finds it doesn't compute the same answer. To prepare for a *p2d2* debugging session, the user has to link his application with a special version of `MPI_Init` which provides the following functionality:

- The process ID of each MPI process is written to a file.
- The program is put into an infinite sleep loop.

The added functionality allows us to attach to the processes of the parallelized version of the code before they progress in their execution.

Now the user starts *p2d2* with the command line:

```
p2d2 -R "mpirun -np 4 P" S
```

which requests that *p2d2* compare the execution of "`mpirun -np 4 P`" with the execution of *S*. After *p2d2* starts up, the following sequence of events occurs. It is depicted in Figure 2.

1. *P2d2* starts a *gdb* to control execution of *S*. Then *p2d2* requests that *gdb* insert a breakpoint at the beginning of process S and at the entry point of function "`__p2d2DiffDetected`". This is the function, discussed in Section 3.2, that gets called when a difference is detected.

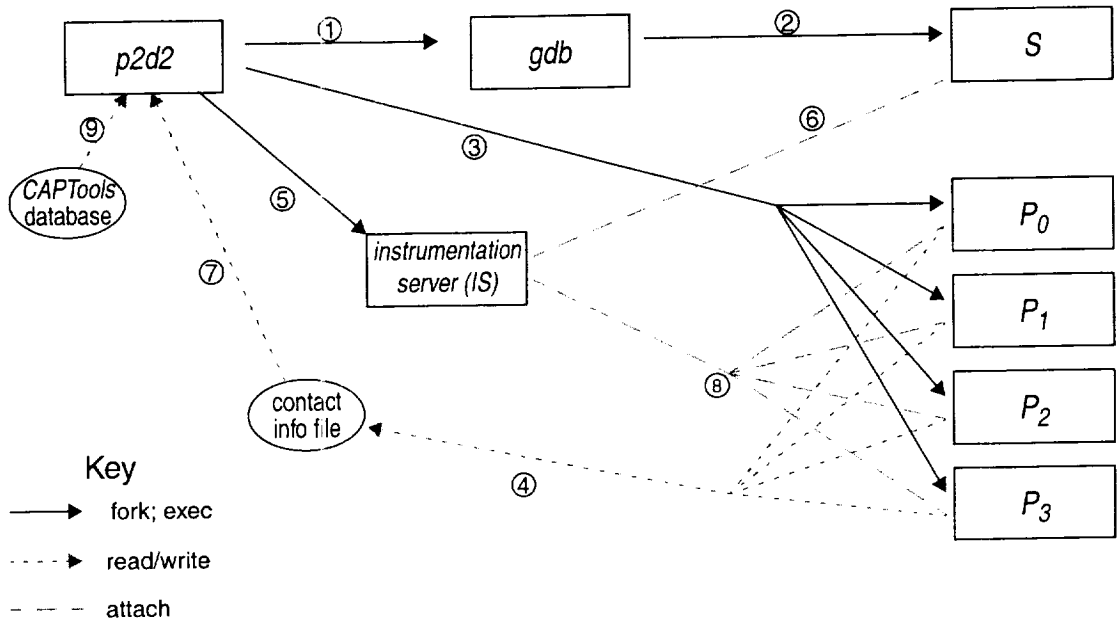

FIGURE 2. Coordination of the comparison activities.

2. The user then selects an array in *p2d2*'s source display and invokes the ***Run*** operation. *P2d2* issues a "`run`" request to the *gdb* controlling *S*.
3. *P2d2* issues the shell command "`mpirun -np 4 P`".
4. The four processes resulting from that command record contact information, including their process ID's, in a file.
5. After it sees that the contact file has been created, *p2d2* starts up *IS*.
6. *IS* attaches to the process running *S*.
7. *P2d2* reads the parallel execution contact information from the file system. It then sends the `attach` requests to *IS*.
8. *IS* attaches to the four processes running *P*.
9. *P2d2* consults the *CAPTools* database and retrieves information about distributed arrays and which functions will need to be instrumented. It also provides the local name in the function of the array that needs to be monitored.

Then *p2d2* and *IS* complete the instrumentation of the processes and proceed with the execution.

- *IS* inserts instrumentation into the process running *S* and the four processes running *P*.
- *IS* detaches from the *P* processes.
- *IS* notifies *p2d2* that the instrumentation of *S* is complete. *P2d2* sends a "continue execution" request to the *gdb* controlling *S*.
- When the inserted instrumentation in *S* and *P* is executed, it establishes communication links between the serial and parallel processes. In our current prototype implementation, we use named pipes for communication.
- At the function entry and exit points that were instrumented, the parallel processes send their state information to the serial process. It compares the parallel data to its own. If there is a difference, it calls `__p2d2DiffDetected`, which causes a trap because of the breakpoint that was set there.

```
      program example
C     main routine
      double precision phi2 (0:101,0:101)
      double precision oldphi2(0:101,0:101)
      read (5,*) n, m
      do i = 1, 43
         ind1 (i) = i + n
         ind2 (i) = i + m
      end do
      call setup_grid (phi2,...)
      do iter = 1, 100
        call copyphi (oldphi2, phi2)
        call update (phi2, oldphi2,...)
      end do
      call output (phi2, nptsx, nptsy)
      return
      end
      subroutine output (phi3,...)
C     Routine that prints the result
      ...
      do j = 0, nptsx+1
         do i = 0, nptsy+1
            phi7 (i,j) = phi3 (i,j)      !Def
         end do
      end do
      write (8,*) phi7
      end do
      return
      end
      subroutine copyphi (oldphi5, phi5)
C     Routine that saves old values
      ...
      do j = 0, 43
        do i = 0, 43
           oldphi5(i,j) = phi5(i,j)      !Def
        end do
      end do
      return
      end
```

```
      subroutine update (phi4, oldphi5, ind1,
     ind2, ...)
C     Routine that updates array
      ...
      do j=0,nptsx+1
        do i=0,nptsy+1
           oldphi4(i,j) = oldphi5(i,j)
        end do
      end do
      do j=1,nptsx
        do i=1,nptsy
           phi4 (i,j) =                   !Def
     #          phi4(i,ind1(j)) +
     #     0.25 * (oldphi4(i, ind2(j)) +
                   oldphi4(i, j+1))
         end do
      end do
      return
      end

      subroutine setup_grid (phi6,...)
C     Routine to set up the initial
C     grid values
      ...
      do j=0,nptsx+1
         do i=0,nptsy+1
           phi6(i,j) = 0.0              !Def
         end do
      end do
      ...
      do j=1,nptsx
         do i=1,nptsy
            phi6(i,j) = 1.0             !Def
         end do
      end do
      return
      end
```

FIGURE 3. Example program source outline.

- When *p2d2* is notified of the trap, it determines that the cause is a difference in state between the serial and parallel executions. It then presents the information to the user.

5 An Example Debugging Session

Suppose a user is parallelizing the Fortran program outlined in Figure 3 with *CAPTools*. During the main iteration loop, the elements of an array are updated by adding a linear combination of values from the previous iteration. The elements used for the update are referenced by indirect addressing. The calculation of the address is determined by the offset values `n` and `m`, which are being read at runtime in the main program (see Figure 3). The user decides to parallelize the code in the second array dimension.

During the parallelization process, the user examines the dependence graph with an eye for removing dependence edges that might result in unnecessary communication or might prevent parallelization altogether. While this gives the user an opportunity to improve code performance, it can also result in incorrectly behaving code.

Figure 4 shows the *CAPTools* display of the dependence edges indicating true dependencies in routine `update`. A true dependence, carried by the second loop index `j` has to be assumed in line 10 of the routine, since the value of the offset `n`

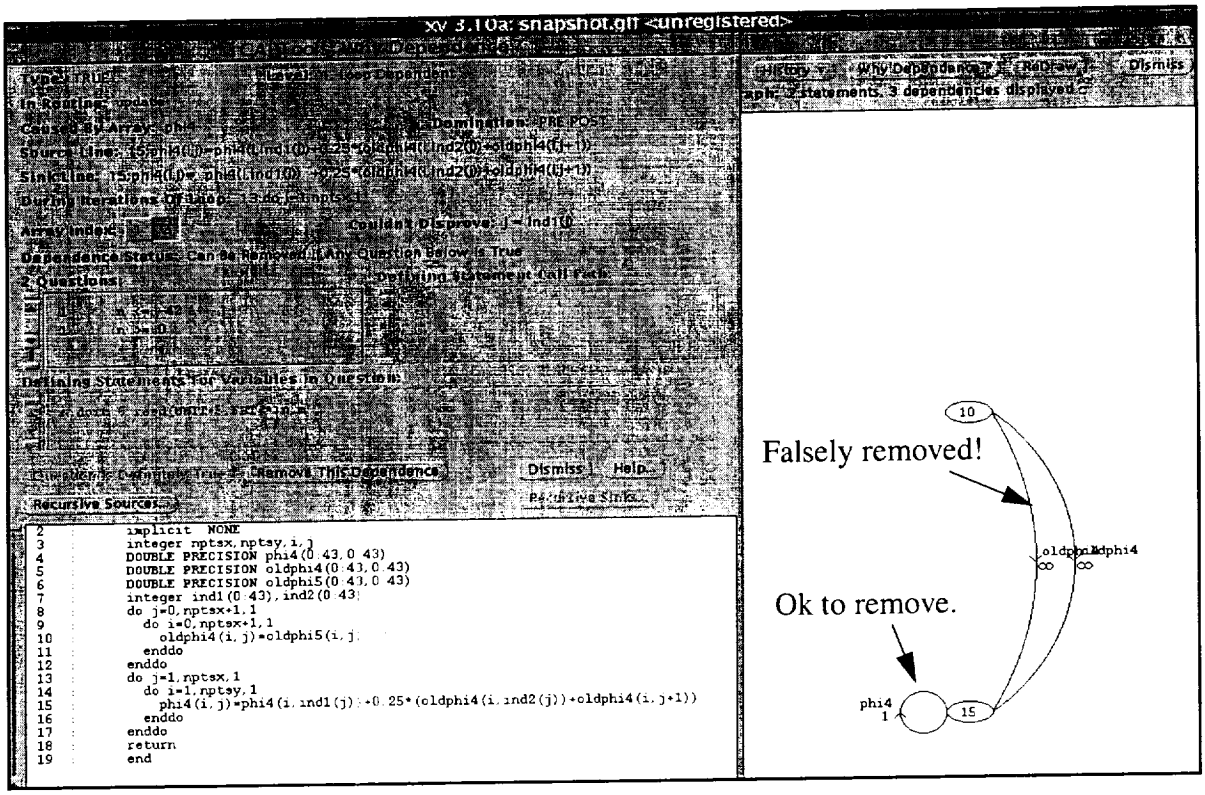

FIGURE 4. Examining and manipulating data dependences with *CAPTools*.

could be negative, but greater than -43. This would lead to a dependence cycle which prevents parallelization of the loop. Let us assume the user has the knowledge that the offset n is greater than or equal to 0. In this case the dependence edge can be removed, enabling parallelization of the loop. Let us further assume that the user, now incorrectly, also removes the edge resulting from the definition of element `oldphi4(i,j)` in the first loop and the reference of `oldphi4(i,ind2(j))` in the second loop. In a real application a user often has to examine dozens of dependencies. In order to achieve a satisfactory level of parallelization, the user will have to remove unnecessary dependence edges, and it is not uncommon to remove the wrong edge. In our example, let us assume that the actual value of `m` is -1. In this case, removing the dependence edge between the statements in line 10 and 15 will result in missing communication of updated values, as we shall see later on.

The user then runs the resulting code (named "`par_test`") and notices that the values of array `phi7` printed in routine `output` are different from those printed by a run of the sequential version. He then invokes the *p2d2* debugger in relative debugging mode with the command:

```
p2d2 -R "mpirun -np 4 par_test" serial_test
```

where "`serial_test`" is the name of the serial executable.

When the *p2d2* display appears, the user brings up a dialog box and asks for value of variable `phi7` in routine `output` to be monitored during execution (see Figure 5). After the user has requested the start of execution, the *CAPTools* database is probed behind the scenes. We have indicated the definitions of suspicious variables in Figure 3 with `!Def`. The probe determines that the following arrays should be checked at entry and exit of the corresponding routines:

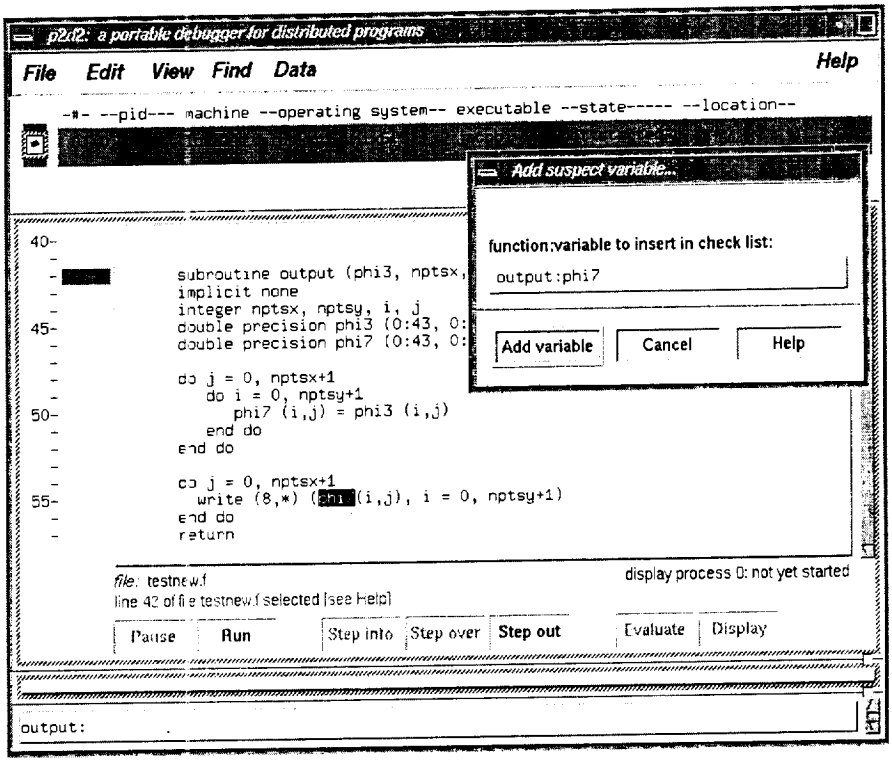

FIGURE 5. Preparing for a relative debugging run in *p2d2*.

- `oldphi5` in `copyphi`
- `phi4` in `update`
- `phi6` in `setup_grid`

After starting both executions, the routines are instrumented and execution continued. The comparison subsequently detects a difference in `phi4` on the exit of `update`. The debugger then displays a message to the user, indicating that a difference was detected in variable `phi4` when exiting subroutine `update` and that the variable had tested equal when entering subroutine `update`. This brackets the error to execution of routine `update`. When the user inspects the parallel version of that routine:

```
      subroutine update(phi4,oldphi5,...)
          ...
      do j=MAX(0,CAP_Lind1),MIN(nptsx+1,CAP_Hind1), 1
        do i=0,nptsx+1,1
           oldphi4(i,j)=oldphi5(i,j)
        enddo
      enddo
      CALL CAP_EXCHANGE(oldphi4(0, CAP_Hind1+1,1,
     #                  oldphi4(0,CAP_Lind1, 1)..., CAP_RIGHT)
      do j=MAX(1,CAP_Lind1),MIN(nptsx,CAP_Hind1),1
        do i= 1,nptsy,1
          phi4(i,j)= phi4(i,ind1(j)) + 0.25*(oldphi4(i,ind2(j))
     #                 +                   oldphi4(i,j+1))
        enddo
      enddo
      return
      END
```

he sees that there is an exchange of the values of `oldphi4` to the right side to obtain the required values of `oldphi4(i,j+1)`, but there is no communication with the left neighbor. Since `ind2(j)=j-1`, stale values of `oldphi4(i,ind2(j))` are used

to update `phi4`. The missing communication routine is due to the erroneous removal of the dependence edge earlier.

6 Prototype Performance Evaluation

The purpose of this section is to evaluate the feasibility of our implementation approach. To quantify the efficiency of our system, we conducted a number of timing experiments comparing the runtime of:

- Instrumented vs. uninstrumented code.
- Hand-instrumented vs. *Dyninst* instrumented code.
- Executables running with our prototype implementation vs. an alternative way to implement relative debugging.

In our prototype implementation we have instrumented the code dynamically using our instrumentation server *IS* (and the *DyninstAPI*). An alternative way to implement our relative debugging system is to run the processes under a debugger such as *gdb* and invoke the comparison routines at breakpoints on entry and exit of subroutine calls. In both cases the comparison routines must be linked with the executable. Using *IS*, the call is actually patched into the code. The application is allowed to run without intervention and the call to the comparison routines will be executed on every entry to and exit from suspicious routines once the target process is continued. By contrast, in the *gdb* approach the target processes trap twice on every instrumented call, at which point *gdb* orchestrates the call to the instrumentation routine and the continuation of execution after that call returns. This potentially causes many context switches.

6.1 Description of the Test Environment

We set up a simple test environment which did not include the *p2d2* debugger. As a sample application, we used an MPI program and compared data between a single and a multiple process MPI run. The single process MPI run is considered the reference version producing the correct results. Communication between the two executables is implemented by using named pipes and is performed by the comparison routines discussed earlier. The single and multiple process executions synchronize after each comparison. In our timings, we measured the elapsed execution time of the reference program. We built single and multiple process executable using the special version of `MPI_Init` which was described in Section 4. The added functionality allows us to attach to the processes with *IS* or with *gdb*, respectively, before they progress in their execution.

In our timing tests we want to insert the function call `compit(arg1)` at the entry and exit point of function `sub1(arg1)`. The routine `compit` performs the data comparison described in Section 3.2. The source code is contained in file `test.f` and the name of the executable is `a.out`. Each approach is implemented using a simple shell script. For both methods the shell script starts the executables and reads the file containing the process IDs. When using dynamic instrumentation we proceed as follows: For each process, we attach with *IS* and issue the sequence of commands as given in Figure 6a.

The first command will break the process out of the infinite loop. The subsequent commands create instrumentation points at entry and exit of subroutine `sub1` and insert calls to the comparison routine `compit`. Then *IS* continues the execution of the executable and detaches.

In the case of using the debugger we attach with *gdb* to each of the processes. Each *gdb* is started with a command file as shown in Figure 6b. The command file breaks the processes out of the infinite loop, sets breakpoints at beginning and end of subroutine `sub1`, and ensures that a call to the comparison routines is executed each time a breakpoint is hit. Note that *gdb*

```
set pid _GO 1
createPoint pid sub1_ ENTRY
insertCall pid compit_ 0
createPoint pid sub1_ EXIT
insertCall pid compit_ 0
continue pid
detach pid
```

FIGURE 6a. *IS* commands

```
gdb commands:
set _GO 1
break test.f: 50        ! entry to sub1
commands
  call compit(arg1)
  continue
end
break test.f:63         ! exit from sub1
commands
  call compit(arg1)
  continue
end
continue
```

FIGURE 6b. *gdb* commands

FIGURE 6. Command sequence for IS and corresponding commands for gdb

does not allow us to set breakpoints at exits of subroutine calls. We therefore need to set the breakpoint at the appropriate line number.

The computationally intensive part of the MPI application is a two dimensional matrix multiplication. We chose an element-by-element comparison test. In our timing experiments we vary the following parameters:

- *NP*: The number of processes in the multiple process execution.
- *NLEN*: The size of the array dimensions in the single process execution.
- *NCOUNT*: The number of times that the suspicious subroutine is being called.
- *NCOMPUTE*: The number of calculations between each comparison.

By varying these four parameters we can investigate the effect on execution time of relative debugging with respect to parallel scalability, and size of the data being compared, as well as frequency and granularity of the required comparisons. Our timings were performed on an SGI Origin2000 with a 400 MHz clock rate. We used *NP*+1 CPUs for our timing experiments.

6.2 Discussion of the timing results

We conducted the following tests:

- Test SSS: Small array, small number of comparisons, small amount of computation.
- Test SLS: Small array, large number of comparisons, small amount of computation.
- Test LSS: Large array, small number of comparisons, small amount of computation.
- Test LSL: Large array, small number of comparisons, large amount of computation.

We compare the runtime of the un-instrumented sequential executable to the execution time of a full relative debugging session. Figure 7 shows the timings for a 16 process parallel execution. Note that the time is given on a logarithmic scale. In

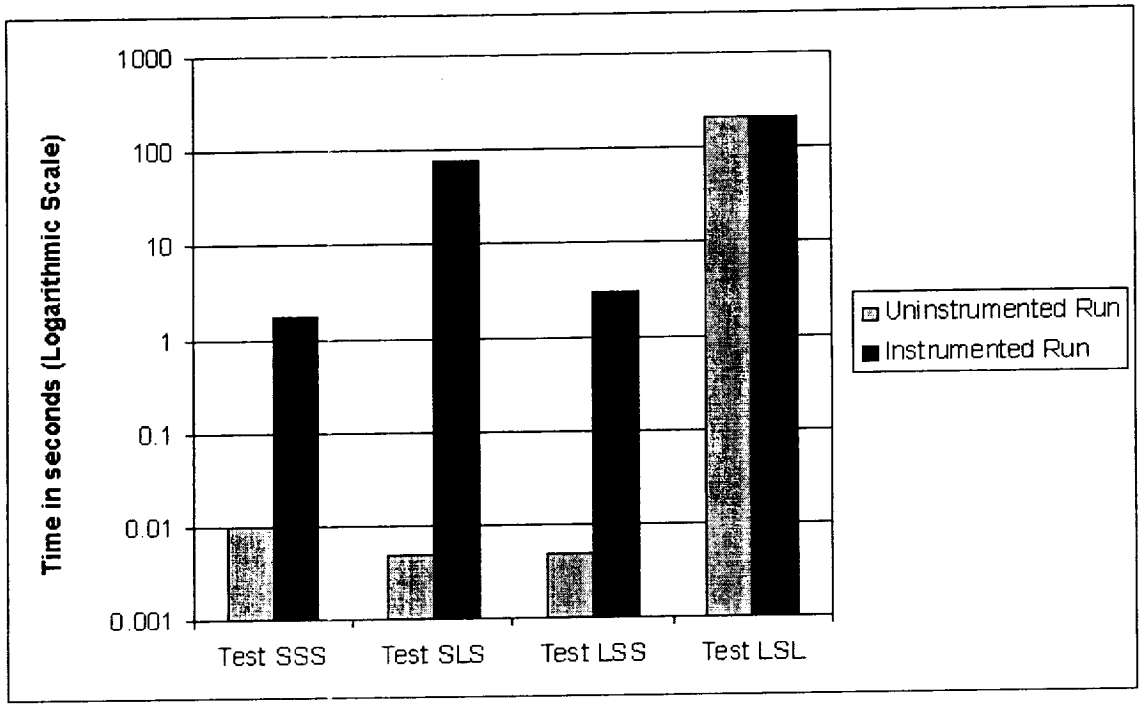

FIGURE 7. Comparison of un-instrumented and instrumented run on 16 processes

tests SSS, SLS, and LSS, the runtime of the un-instrumented program takes only a fraction of a second and increases to up to a minute during a relative debugging session. These tests correspond to fine- grained checking with little computation being performed between the comparisons. For test LSL, which corresponds to a coarse-grained instrumentation, the increase in run timeis not as extreme. The timings show that a full relative debugging session could can take a long time depending on the runtime of the un-instrumented code, the size of the data that needs to be checked, and the granularity of the checks. The advantage of our method is, that once the user has set up the relative debugging run by indicating an initial suspect variable, the debugging session can run in batch mode, without any further user interaction. While the *p2d2* debugger, and thus our prototype, currently do not support batch processing, the timings suggest strongly that such an interface is necessary.

Even when running in batch mode, care has to be taken when fine-grained comparison is being used. A completely instrumented program making fine-grained comparisons could take a very long time to run. It would probably be faster to make multiple runs of the code with the instrumentation points changing to narrow down the first difference. In the future work section below (Section 9) we discuss such a possibility.

In Figure 8 we compare the runtime of dynamically instrumented executables to that of executables whose source code had been instrumented and re-compiled. Except for test SSS on few processes, where the execution time is extremely short, we found that there was basically not much difference in the runtime.

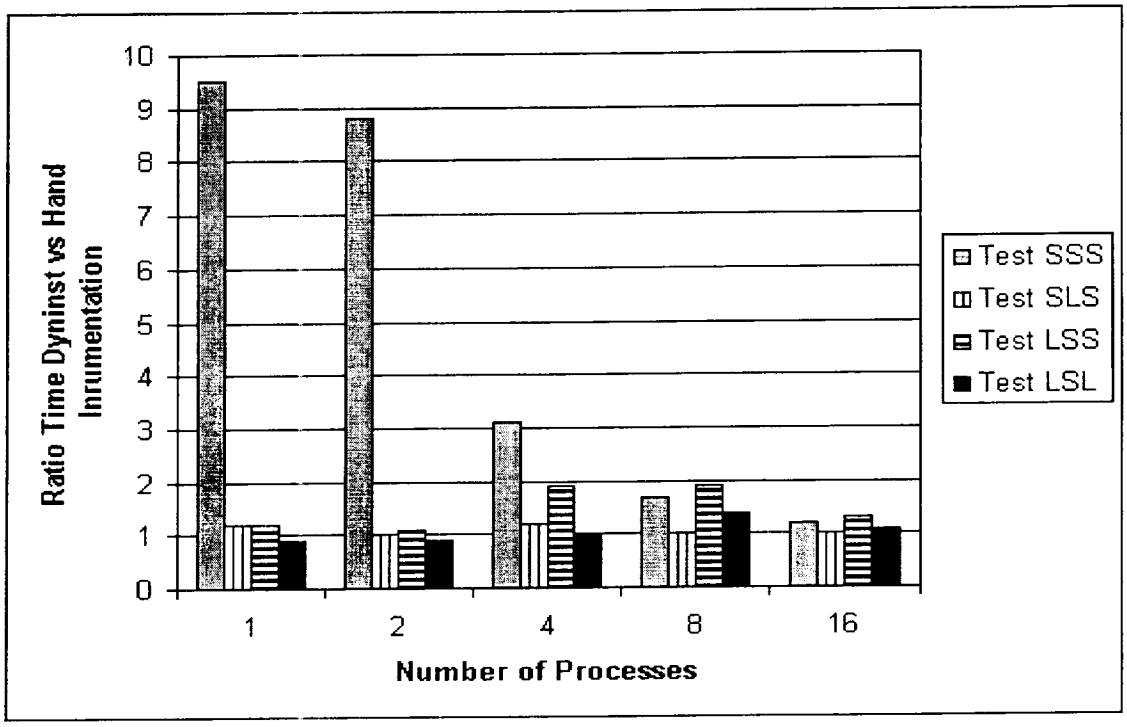

FIGURE 8. Runtime comparison of Dyninst vs. hand instrumented executables

Figure 9 shows the ratio of the runtime of relative debugging sessions using our prototype system versus the gdb-based approach. The timings show that the *gdb*-based approach has a more severe impact on the execution time than using dynamic instrumentation, particularly when the frequency of the comparisons performed is high. The timings indicate that the strength of dynamic instrumentation, when it comes to relative debugging, is that it imposes little overhead on execution time when small amounts of data are compared many times. This corresponds to a fine-grained instrumentation on statement level. Unfortunately, at the time that this study was done, the implementation of *Dyninst* for the SGI did not allow for arbitrary instrumentation, but only at entry and exit of subroutine calls. We will discuss some of the challenges of fine-grained instrumentation in Section 9.

7 Implementation experiences

While we see great promise in the progress to date toward our goal of automatic support for debugging tool-parallelized programs, we have also observed limitations. Many of these restrictions are imposed by the foundation software that we have used to build our implementation. For example, the currently released version of *CAPTools* does not provide information in its database about arrays distributed across more than one dimension, since this information is not relevant for the typical use of CAPTools. However, this is not an inherent restriction in CAPTools, and the developers can provide us with a special version of the database that will allow us to retrieve this kind of information.

In the case of *Dyninst*, the implementations for the platforms we tested are restricted in several ways. Perhaps the most significant is that instrumentation can currently only be placed at subroutine entry and exit. It is our understanding that,

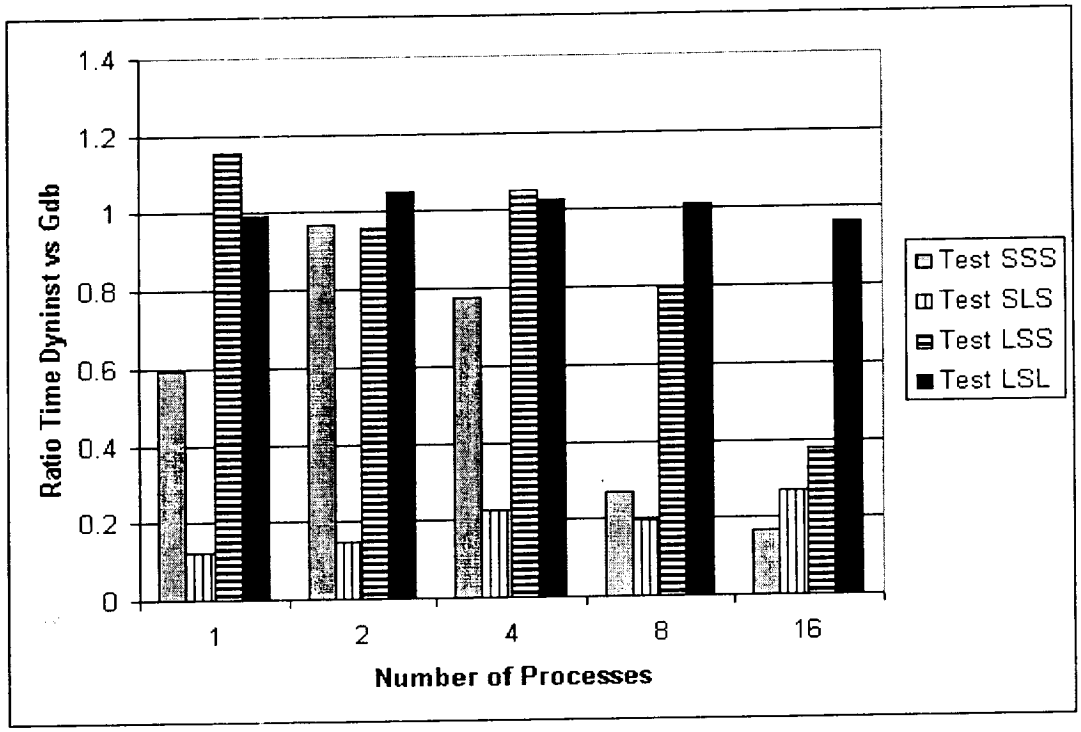

FIGURE 9. Runtime comparison of IS vs. gdb-based relative debugging.

eventually, the package will permit instrumentation at arbitrary instructions in the code, effectively removing this restriction. In addition to this limitation, code patched in by the version of the *DyninstAPI* that we are using is unable to access function parameters in the ninth position or thereafter. This problem is particularly felt in Fortran codes, where long parameter lists are common. *Dyninst* also has a limited knowledge of the symbol table. In particular, it knows the location of global variables, but not locals or parameters.

We can get around some of the *Dyninst* symbol table limitations by using *gdb* to get that information. Unfortunately, operating system issues come up when both our *Dyninst*-based instrumentation server and *gdb* want to attach to the same process. On some systems, such as Linux, only one can be attached at a time. In that case, our implementation will need to coordinate attach and detach requests. Our experience on Linux shows, however, that a process cannot successfully be attached by *Dyninst* after it has been detached. For the purposes of the prototype, we restricted ourselves to an IRIX implementation where both *gdb* and our *Dyninst*-based instrumentation server could be attached at the same time.

Other issues also arise as a result of trying to debug *Dyninst*-instrumented codes. For example, when execution stops in a routine called from an instrumentation point, the runtime stack is in a state that *gdb* cannot handle—there is a return address on the stack that is outside the range that *gdb* is looking for.

In addition to limitations in the software packages used by the prototype, we should also point out some within the prototype itself. In particular, the current implementation requires the target executables to include the instrumentation routines described in Section 3.2. We plan to use dynamic linking in the future to address this restriction.

One additional limitation of our prototype is that, while comparing executions we currently require that the checkpoints to be compared occur in the same order in the two runs. In the future we can address this restriction in a manner similar to *Guard* [2] by saving out-of-sequence checkpoints in the file system until the comparison can be made.

8 Related Work

Guard [1][2][3] is a relative debugger for parallel programs developed at the Griffith University in Brisbane Australia. In contrast to our approach, where two executables communicate data directly with each other and do the comparison, in *Guard* the debugger collects the data from the executables and does the comparison. Also, *Guard* does not aim particularly at automatically parallelized programs. Information about where to do the comparisons and which parts of the data to compare are provided by the user via the command language. To compare array data from parallel programs, the user must describe the decomposition manually using a distributed array syntax.

In other work on debugging automatically parallelized programs, Cohn [6] has investigated having the debugger provide a sequential view of an executing parallel program. While he does not use relative debugging techniques, his analysis of consistency issues between sequential and parallel executions could be useful in identifying candidate instrumentation points for making comparisons in the relative debugging approach.

The idea of using information from parallelization tools to aid in debugging has also been around for some time. For example, Hood, Kennedy, and Mellor-Crummey [13] used dependence information from a parallelizing compiler to determine which data accesses to instrument to find races in a shared-memory program execution.

9 Project Status and Future Work

We have built a prototype of a relative debugging system for comparing serial codes and their tool-produced parallel counterparts where array comparisons are done either by computing checksums or by doing element-by-element equality tests. After the user specifies a variable and scope to be checked, the debugger uses the *CAPTools* database to determine which variables should be monitored and in which functions. We used the dynamic instrumentation tool *Dyninst* in order to minimize the overhead involved in making the comparisons. We ran extensive timings and tested the need in such an environment for dynamically inserted procedure calls versus interpreted calls.

In the near future we will integrate the relative debugging features more seamless into *p2d2*. In particular, we would like to have debugging requests that the user makes on the serial code, also be performed on the parallel version. In order to do that, we need to modify the *p2d2* user interface to support multiple computations executing simultaneously. In addition, we must get *CAPTools* to provide information about how the serial program was transformed into its parallel form. This will permit us to determine places in the code where there should be consistency between states in sequential and parallel versions.

Furthermore, while *CAPTools* allows for cyclic and block-cyclic array distributions, we currently support only blockwise distributions. In the future we will address this issue.

Our approach for relative debugging of tool-parallelized distributed memory codes will also work for shared memory codes parallelized with tool support. In the near future we will extend our prototype to work with codes produced by *CAPO* [5] which is based on *CAPTools* and produces OpenMP [18] codes.

At the moment our system only helps to narrow down the error location on a subroutine level. The user still has to has examine potentially large sections of automatically generated code. In the longer term, we would like to use intraprocedural dataflow information from *CAPTools* in order to pinpoint execution differences to particular statements, rather than procedure bodies. We recently began working with Steve Johnson, of the University of Greenwich, on an implementation that uses a combination of state examination and re-execution to backtrack detected differences to the first place where they occur. In this approach instrumentation is inserted at USE points of variables to compare values across programs. When a difference is detected, the debugger uses information from *CAPTools* to find the possible definition points of the variable. If the values used on the right-hand-sides of the definition points are still live, the debugger checks for differences in them. If a difference is detected, the process is repeated looking at the definition points of the new difference variable. If the values of the right-hand-side variables have been overwritten by subsequent execution, then these USEs are instrumented and the program is rerun. Viewed another way, we are essentially using program slice information [14] to help minimize the number of instrumentation points, and we are using liveness information to help minimize the number of re-executions. Our expectation is that this approach will automatically isolate bugs with great precision.

In our experience, the majority of errors are due to incorrect information provided by the user. In further development of this work, CAPTools could store information about deleted dependencies and user information in its database. This would allow the debugging algorithm to investigate if a deleted dependence or other user provided information could potentially be responsible for the incorrect parallel execution. As a first step, a list of possible problems could be displayed to the user who will then have the opportunity check his assumptions.

Besides the relative debugging work, we would also like to experiment with other uses for dynamic instrumentation in debugging. For example, we would like to use *Dyninst* to provide fast conditional breakpoints in *p2d2*.

10 Conclusions

In this paper we have described a system that simplifies the process of debugging programs produced by computer-aided parallelization tools. The system uses relative debugging techniques to compare serial and parallel executions in order to show where the computations begin to differ. It uses information produced by the parallelization tool to drive the comparison process without user intervention. In addition, the use of dynamic instrumentation makes the comparisons efficient. We feel that this approach holds great promise for meeting the goal of providing automated support for isolating bugs introduced in the parallelization process.

Acknowledgments

The authors would like to thank Henry Jin and Rob Van der Wijngaart of NAS for their comments on this paper. Henry was also particularly helpful as a local expert on *CAPTools*. We also thank Steve Johnson and Peter Leggett of the University of Greenwich who were very responsive in providing routines to retrieve interprocedural information from the *CAPTools*

database. Ravi Samtaney from the California Institute of Technology provided a sequential version of the RM3d code for the solution of Euler's equations in three dimensions, which we used as a test program. We also thank Randy Kaemmerer of NASA Ames for his careful reading of the paper and the suggestions he made for improving it.

References

[1] Abramson, D., Foster, I., Michalakes, J., and Sosic, R. 1996. "Relative Debugging: A New Methodology for Debugging Scientific Applications." *Communications of the ACM 39* (11).

[2] Abramson, D., Sosic, R., and Watson, G. 1996. "Implementation Techniques for a Parallel Relative Debugger." *Proceedings of PACT'96*, Boston.

[3] Abramson, D. and Watson, G. 1997. "Relative Debugging for Parallel Systems." *Proceedings of PCW 97*, Canberra, Australia.

[4] Barker, D. P. 1994. Personal Communication.

[5] CAPTools-based Automatic Parallelizer with OpenMP (CAPO). `http://www.nas.nasa.gov/~hjin/CAPO.html`.

[6] Cohn, R. 1991."Source Level Debugging of Automatically Parallelized Code." *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*.

[7] Computer Aided Parallelization Tools (CAPTools). `http://captools.gre.ac.uk/`.

[8] The Dyninst API. `http://www.cs.umd.edu/projects/dyninstAPI/`.

[9] The Free Software Foundation. `http://www.fsf.org/`.

[10] Hood, R. 1996. "The *p2d2* Project: Building a Portable Distributed Debugger." *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA.

[11] Hood, R. and Jost, G. 2000. "A Debugger for Computational Grid Applications." *Proceedings of the Heterogeneous Computing Workshop*, Cancun, Mexico.

[12] Hood, R. and Jost, G. 2000. "Support for Debugging Automatically Parallelized Programs." *Proceedings of AADEBUG 2000*, Munich, Germany.

[13] Hood, R., Kennedy, K., and Mellor-Crummey, J. 1990. "Parallel Program Debugging with On-the-fly Anomaly Detection." *Proceedings of Supercomputing'90*, New York.

[14] Horwitz, S., Reps, T., and Binkley, D. 1990."Interprocedural Slicing Using Dependence Graphs." *ACM Transactions on Programming Languages and Systems 12* (1).

[15] Johnson, S.P., Cross, M., Everett, M. G. 1996."Exploitation of Symbolic Information in Interprocedural Dependence Analysis", *Parallel Computing 22*:197-226

[16] Kessler, P. 1990."Fast Breakpoints: Design and Implementation." *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY.

[17] Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[18] OpenMP. `http://www.openmp.org/`.

[19] The *p2d2* Project. `http://www.nas.nasa.gov/Tools/p2d2`.

[20] Sunderam, V. 1990. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience* 2(4):315-339, 1990.

[21] Wolfe, M. 1989.*Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, MA.

Contact Author

Gabriele Jost

NASA Ames Research Center, M/S T27A-2
Moffett Field, CA 94035-1000
Tel (650) 604 4431
Fax (650) 604 3957

gjost@nas.nasa.gov

Keywords:

Relative debugging, message passing programs, automatic parallelization, dynamic instrumentation.